GDB, Emacs и Google Summer of Code

Дмитрий Джус

МГТУ им. Н.Э.Баумана

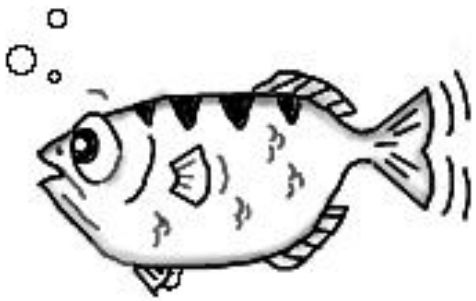
2009

План

Что такое отладчик

- Debugger помогает избавиться от багов
- Стандартные функции отладчика
  - Пошаговое исполнение программы
  - Условный останов программы
  - Просмотр и изменение значений переменных, памяти, регистров
  - Изучение стека, нитей и дизассемблированного кода

GNU Debugger — свободный отладчик



- Переносимый отладчик для различных языков, включая C, C++ и Fortran
- Разработка начата rms в 1986 году в рамках проекта GNU и ныне продолжается силами сообщества при поддержке крупных компаний
- Стандартный отладчик для многих современных Unix-систем
- Используется в качестве компонента для ряда интегрированных сред разработки
- Поддержка удалённой отладки
- Различные интерфейсы (CLI для ручной работы, Machine Interface для IDE)

Новые возможности в GDB 7.0

- Безостановочная отладка многопоточных приложений
- Поддержка Python в качестве языка расширения
- Обратимая отладка

Зачем нужна безостановочная отладка

- Многопоточные приложения — тренд
- Нужно обеспечить недеструктивный контроль: отлаживать только ту нить, в которой проблемы

pthread.c

- Запускаем пять нитей, одна из которых выполняет `f2()`

```
for (i = 0; i < 5; i++)
{
    pthread_create(&threads[i],
                   NULL,
                   ((i + 1) % 5) ? f1 : f2,
                   (void *)i);
}
```

- `f1()` и `f2()` — рабочие функции; `f2()` имеет проблему

```
void *f1(void *tid) { while(1); }
void *f2(void *tid) { /* bug here */ }
```

Как было раньше

```
(gdb)
```

Как было раньше

```
(gdb) break f2
(gdb)
```

Как было раньше

```
(gdb) break f2
(gdb) run
…
Breakpoint 1, f2 (tid=0x0) at pthread.c:5
5 void *f2(void *tid) { /* bug here */ }
(gdb)
```

Как было раньше

```
(gdb) break f2
(gdb) run
…
Breakpoint 1, f2 (tid=0x0) at pthread.c:5
5 void *f2(void *tid) { /* bug here */ }
(gdb) info thread
* 6   f2 (tid=0x0) at pthread.c:5
  5   f1 (tid=0x0) at pthread.c:4
  4   f1 (tid=0x0) at pthread.c:4
  3   f1 (tid=0x0) at pthread.c:4
  2   f1 (tid=0x0) at pthread.c:4
  1   in __kernel_vsyscall ()
(gdb)
```

Как стало

```
(gdb)
```

Как стало

```
(gdb) set non-stop on
(gdb)
```

Как стало

```
(gdb) set non-stop on
(gdb) set target-async on
(gdb)
```

Как стало

```
(gdb) set non-stop on
(gdb) set target-async on
(gdb) break f2
(gdb)
```

Как стало

```
(gdb) set non-stop on
(gdb) set target-async on
(gdb) break f2
(gdb) run
…
Breakpoint 1, f2 (tid=0x0) at pthread.c:5
5 void *f2(void *tid) { /* bug here */ }
(gdb)
```

Как стало

```
(gdb) set non-stop on
(gdb) set target-async on
(gdb) break f2
(gdb) run
…
Breakpoint 1, f2 (tid=0x0) at pthread.c:5
5 void *f2(void *tid) { /* bug here */ }
(gdb) info thread
  6 f2 (tid=0x0) at pthread.c:5
  5 (running)
  4 (running)
  3 (running)
  2 (running)
* 1 (running)
(gdb)
```

Как стало

```
(gdb) set non-stop on
(gdb) set target-async on
(gdb) break f2
(gdb) run
…
Breakpoint 1, f2 (tid=0x0) at pthread.c:5
5 void *f2(void *tid) { /* bug here */ }
(gdb) info thread
  6 f2 (tid=0x0) at pthread.c:5
  5 (running)
  4 (running)
  3 (running)
  2 (running)
* 1 (running)
(gdb)
```

Асинхронность

- В безостановочном режиме управляющие команды (`continue`) относятся только к текущей нити
- Для раздельного управления нитями нужна возможность посылать команды, когда программа выполняется — знак &

```
(gdb)
```

Асинхронность

- В безостановочном режиме управляющие команды (`continue`) относятся только к текущей нити
- Для раздельного управления нитями нужна возможность посылать команды, когда программа выполняется — знак &

```
(gdb) info thread
  3 f2 (tid=0x4) at pthread.c:5
* 2 f1 (tid=0x0) at pthread.c:4
  1 (running)
(gdb)
```

Асинхронность

- В безостановочном режиме управляющие команды (`continue`) относятся только к текущей нити
- Для раздельного управления нитями нужна возможность посылать команды, когда программа выполняется — знак &

```
(gdb) info thread
  3 f2 (tid=0x4) at pthread.c:5
* 2 f1 (tid=0x0) at pthread.c:4
  1 (running)
(gdb) continue &
Continuing.
(gdb)
```

Асинхронность

- В безостановочном режиме управляющие команды (`continue`) относятся только к текущей нити
- Для раздельного управления нитями нужна возможность посылать команды, когда программа выполняется — знак &

```
(gdb) info thread
  3 f2 (tid=0x4) at pthread.c:5
* 2 f1 (tid=0x0) at pthread.c:4
  1 (running)
(gdb) continue &
Continuing.
(gdb) info thread
  3 f2 (tid=0x4) at pthread.c:5
* 2 (running)
  1 (running)
```

Поддержка со стороны IDE

- При безостановочной отладке обмен данными с GDB уже не синхронный («запрос — ответ»)
- Асинхронный стиль работы требует от клиента готовности в любой момент реагировать на поступающие от отладчика уведомления о событиях

Зачем встраивать Python в отладчик

- В процессе работы возникает необходимость расширять функционал отладчика
- Стандартный язык сценариев GDB примитивен и ограничен

Как работает поддержка Python

- Новая команда `python` для выполнения Python-кода внутри отладчика
- Python-модуль `gdb` предоставляет доступ к информации и объектам отлаживаемой программы
- Описанные на Python функции можно вызывать из GDB

Простая задача для Python API

- По умолчанию STL-контейнеры выводятся в GDB в малочитабельном виде

```
std::list<std::string> l;
std::string s = "foobar";
l.push_back(s);
```

```
(gdb) print s
$1 = {static npos=4294967295,
_M_dataplus={<std::allocator<char>>=
 {<__gnu_cxx::new_allocator<char>>=
  {<No data fields>}, <No data fields>},
_M_p=0x804b014 "foobar"}}
```

- Не STL'ем единым

Создание новых функций для отладчика

- Наследованный от `gdb.Function` класс описывает функцию

```
class stl_str(gdb.Function):
  def __init__(self):
    gdb.Function.__init__(self, "stl_str")

  def invoke (self, val):
    return val['_M_dataplus']['_M_p'].string()
stl_str()
```

- Новая функция доступна из GDB

```
(gdb) print $stl_str(s)
$1 = "foobar"
```

- Созданную функцию приходится вызывать явно

Pretty-printing

- Библиотеки (libstdc++, Glib) могут устанавливать в систему Python-модули, которые выполняют отображение сложных структур данных
- Соответствующие объектным файлам отлаживаемой программы модули подгружаются в GDB
- Визуализаторы *автоматически* используются для переменных подходящего типа

```
(gdb) print s
$1 = "foobar"
(gdb) print l
$2 = std::list = {
  [0] = "foobar"
}
```

- Одинаково работает в CLI & MI

Расширение программ языками высокого уровня

- Что дала поддержка Python в GDB:
  - Использование распространённого языка для развития функций отладчика
  - Поддержка визуализации сложных структур данных
  - Другие, ещё не исследованные приложения
- Подход применяется со времён Emacs: наличие в программе гибкого и выразительного языка расширения позволяет быстрее и проще расширять её возможности
- Другие примеры: Mozilla (XUL), Gimp (Scheme)

Курс на IDE

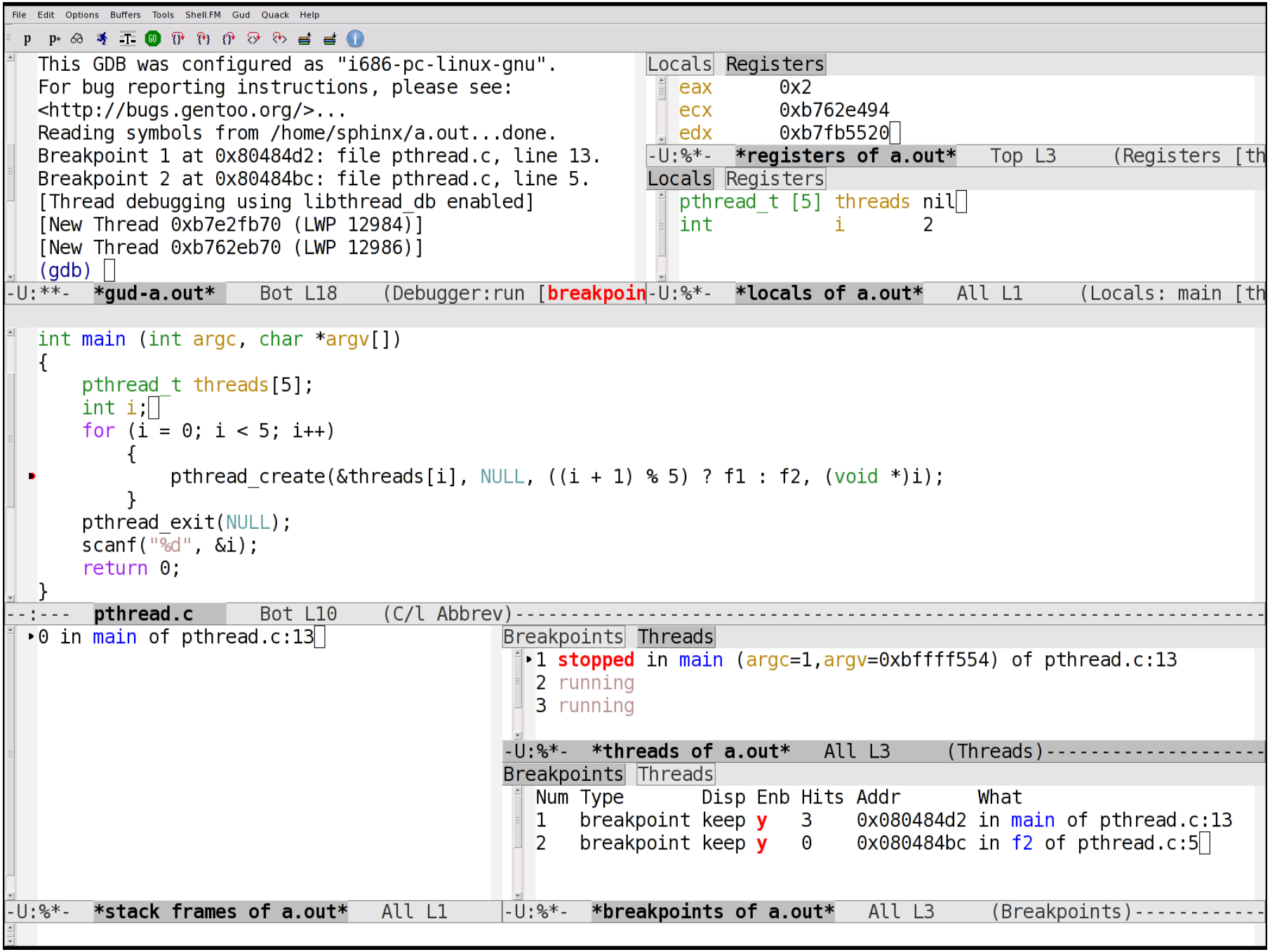
- Новые возможности требуют поддержки со стороны IDE
  - Интерфейс для безостановочной отладки
  - Pretty-printing должен быть явно включен
- Безостановочная отладка уже поддерживается Eclipse и Emacs

GNU Emacs

- Расширяемая настраиваемая самодокументируемая операционная среда
- Компактное переносимое ядро на C
- Большая часть функций описана на Emacs Lisp
- Широкие возможности по взаимодействию с окружающей средой (сеть, IPC)
- Emacs — преимущественно текстовая среда

Emacs как IDE

- GDB запускается как подчинённый процесс
- Обмен командами идёт с помощью Machine Interface
- С различных буферах показана информация от отладчика: стек, точки останова, нити (с возможностью управления), локальные переменные и регистры
- Дополнительно открываются буферы с дизассемблированным кодом и памятью


File Edit Options Buffers Tools Shell.FM Gud Quack Help
This GDB was configured as "i686-pc-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.gentoo.org/>...
Reading symbols from /home/sphinx/a.out...done.
Breakpoint 1 at 0x80484d2: file pthread.c, line 13.
Breakpoint 2 at 0x80484bc: file pthread.c, line 5.
[Thread debugging using libthread_db enabled]
[New Thread 0xb7e2fb70 (LWP 12984)]
[New Thread 0xb762eb70 (LWP 12986)]
(gdb)
-U:**- *gud-a.out* Bot L18 (Debugger:run [breakpoin
Locals Registers
eax 0x2
ecx 0xb762e494
edx 0xb7fb5520
-U:%*- *registers of a.out* Top L3 (Registers [th
Locals Registers
pthread_t [5] threads nil
int i 2
-U:%*- *locals of a.out* All L1 (Locals: main [th
int main (int argc, char *argv[])
{
pthread_t threads[5];
int i;
for (i = 0; i < 5; i++)
{
pthread_create(&threads[i], NULL, ((i + 1) % 5) ? f1 : f2, (void *)i);
}
pthread_exit(NULL);
scanf("%d", &i);
return 0;
}
--:--- pthread.c Bot L10 (C/l Abbrev)
0 in main of pthread.c:13
-U:%*- *stack frames of a.out* All L1
Breakpoints Threads
1 stopped in main (argc=1,argv=0xbffff554) of pthread.c:13
2 running
3 running
-U:%*- *threads of a.out* All L3 (Threads)
Breakpoints Threads
Num Type Disp Enb Hits Addr What
1 breakpoint keep y 3 0x080484d2 in main of pthread.c:13
2 breakpoint keep y 0 0x080484bc in f2 of pthread.c:5
-U:%*- *breakpoints of a.out* All L3 (Breakpoints)

GDB в составе IDE

- При работе через Machine Interface (`gdb -i=mi`) информация выдаётся в структурированном виде

```
(gdb) -stack-info-frame
^done,frame={level="0",
             addr="0x0804888f",func="main",
             file="stl.c",
             fullname="/home/dzhus/stl.c",
             line="10"}
```

- GDB уведомляет клиента об изменениях состояния

```
=thread-created,id="6",group-id="22226"
*running,thread-id="6"
```

Чего не хватало в Emacs

- Ранее Emacs использовал устарейвший командный интерфейс «с аннотациями»
- Не было поддержки безостановочной отладки
- Кто, если не мы?

Google Summer of Code

- Международная летняя программа для студентов
- Под руководством ментора из числа опытных разработчиков студент работает над проектом в области свободного ПО
- С 2005 года в программе успешно приняли участие 2500 студентов из 98 стран

Как устроена программа GSoC

- В начале года *организации* подают заявки на участие, на основе которых их принимают в программу и распределяют слоты для участников
- В марте *студенты* подают заявки с описанием своих *проектов*, которые рассматриваются организациями
- На основе заявок определяется список студентов, которые будут участвовать в программе, назначаются *менторы*
- Основная работа начинается в конце мая
- В середине июля — *смотр*
- Проекты проходят *финальный смотр* в середине августа

Основные достижения

- Emacs-интерфейс к GDB полностью переведён на MI
- За счёт перехода на MI теперь поддерживается безостановочная отладка и просмотр разных нитей/фреймов одновременно
- Множественные улучшения и рефакторинг кода
  - JSON-парсер вместо регулярных выражений
  - Снижено связывание за счёт использования модели с оповещениями для обновления информации в буферах IDE

«Нет» регулярным выражениям

- JSON — простой и гибкий формат обмена структурированными данными (объекты с полями + списки)
- Формат ответов MI хорошо ложится на модель JSON
  - **MI**

```
locals=[{name="threads", type="pthread_t␣[5]"},
        {name="i", type="int",value="3"}]
```

  - **JSON**

```
{"locals":[{"name":"threads",
            "type":"pthread_t␣[5]"},
           {"name":"i",
            "type":"int",
            "value":"3"}]}
```

- Использовать рег. выражения для разбора MI неудобно и неправильно (КС-грамматика)

MI в Lisp

- MI тривиально переводится в JSON, после чего разбирается JSON-парсером
  - **MI**

```
locals=[{name="threads", type="pthread_t␣[5]"},
        {name="i", type="int",value="3"}]
```

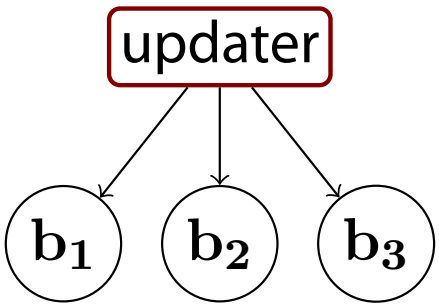
  - **Lisp**

```
((locals .
         [((type . "pthread_t␣[5]")
           (name . "threads"))
          ((value . "3")
           (type . "int")
           (name . "i"))]))
```

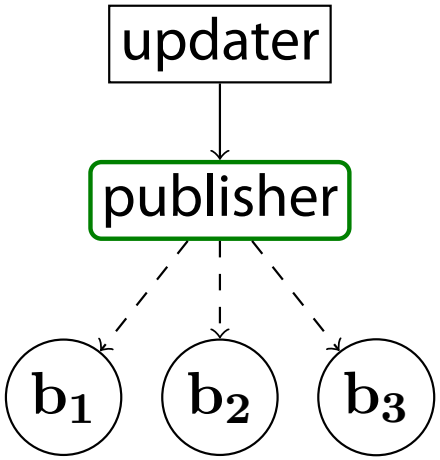
- Работать с естественными Лисп-структурами проще, чем с результатами применения регулярных выражений

Pub & sub

- Буфер каждого типа обновлялся явным образом: расходы на поиск и недостаточная гибкость



Было



Стало

```
(defun gdb-update ()
  (update-locals)
  (update-registers)
  ...)
```

```
(defun gdb-update ()
  (emit-signal
   buf-publisher
   'update))
```

Опыт GSoC

- Обсуждения в рассылках для разработчиков GDB и Emacs и в `comp.lang.lisp`
- Прямой доступ к Emacs CVS
- Рабочий процесс: личный Mercurial плюс upstream CVS.
- Потребовалось больше рефакторинга, чем казалось сначала
- Объём выполненных работ:
  - 285 коммитов
  - Общие изменения 6915 (+), 3994 (-)
  - Итоговые изменения 2390 (+), 802 (-)

Tips & Tricks

- Начинать общаться со своей организацией *до* начала программы
- Заявка на участие должна отражать готовность работать
- План работ помогает разобраться в задаче
- Постоянная работа с собственными мыслями и наблюдениями
- Взаимодействие с сообществом!
- Это *не* соревнование

О чём был доклад

- Свободный отладчик GDB развивается и отвечает на вызовы времени
- Лето с Google — прекрасная возможность получить опыт и принести пользу сообществу и себе